Instructions for L90 Practical: Sentiment Detection of Reviews

Simone Teufel, Adrian Scoica, Yiannos Stathopoulos

October 28, 2016

This practical concerns sentiment classification of movie reviews using a machine learning approach based on bag-of-word features, using a sentiment lexicon, a stemmer, a POS-tagger and a syntactic parser. You must use the MPhil machines for this task. We provide your own personal VM on these machines. Do not use any other packages than those described below.

You will find 1000 positive and 1000 negative movie reviews in `/usr/groups/mphil/L90/data/{pos,neg}`. To prepare yourself for this practical, you should have a look at a few of these texts; you will write a program that decides whether a random unseen movie review is positive or negative, and an assessed report in the form of a scientific article that describes the results you achieved.

Advice: Please read through the entire instruction sheet and familiarise yourself with all requirements before you start coding or otherwise solving the tasks. Writing clean, modular code can make the difference between solving the assignment in a matter of hours, and taking days to run all experiments.

Note: You will be asked to submit the source code used in solving the questions in this practical when submitting your final report.

1 Part One: Baseline and Essentials

How could one automatically classify movie reviews according to their sentiment? Your task in Part One is to establish two commonly used baselines – by implementing and evaluating several NLP methods on this task. Part One will be covered in the demonstrated session on 14/11. After this session, you should have a baseline system, or know what you still have to do to build one.

1.1 Symbolic approach – sentiment lexicon

If we had access to a sentiment lexicon, then there are ways to solve the problem without Machine Learning. One might simply look up every open-class word in the lexicon, and compute a binary score S_{binary} by counting how many words match either a positive, or a negative word entry in the sentiment lexicon $SLex$.

$$S_{binary}(w_1 w_2 ... w_n) = \sum_{i=1}^{n} \text{sgn}(SLex[w_i])$$

If the sentiment lexicon also has information about the magnitude of sentiment (eg. *"excellent"* would have higher magnitude than *"good"*), we could take a more fine-grained approach by adding up all sentiment scores, and deciding the polarity of the movie review using the sign of the weighted score $S_{weighted}$.

$$S_{weighted}(w_1 w_2 ... w_n) = \sum_{i=1}^{n} SLex[w_i]$$

(If you don't know a method of estimating these weights automatically (and we don't expect you to), then choose the weights intuitively *once* before running the experiment.)

Your first task is to implement these approaches using the sentiment lexicon in `/usr/groups/mphil/l90/resources/sent_lexicon`, which was taken from the following work:

> *Theresa Wilson, Janyce Wiebe, and Paul Hoffmann (2005). Recognizing Contextual Polarity in Phrase-Level Sentiment Analysis. Proc. of HLT-EMNLP-2005.*

Their lexicon also records the magnitude of sentiment, so you can implement both the binary and the weighted solutions (please use a switch in your program).

1.2 Tokenisation

You will now need to *tokenize* the text. Don't use an off-the shelf tokeniser, but do it yourself. The two most important rules are to (generally) split on whitespace, and to split punctuation off into their own tokens. Other than that, you will need to find an acceptable treatment for mid-word alphanumerics (eight-year-old-child), contractions (I'll, don't) and genitives (Paul's). And many more but we're asking you to do this task fast. Your guiding principle should be that the tokens you produce should be good meaning-encapsulating units for the task at hand. Similar meanings across different ways of writing should result in similar representations.

Don't spend too much time on this (relatively boring, but important) task. Apply the 80–20 (covering 80% of results in the first 20% of time is often enough). Use the data and your intuition to decide which actions you should do first to achieve this. While you are coding this, try to think of a good, succinct way of explaining your treatmend for the report.

1.3 Answering questions in statistically significant ways

Having implemented both lexicon methods above, consider answering the following question:

> (Q0.1) Does using the magnitude improve results?

Oftentimes, answering questions like this about the performance of different signals and/or algorithms by simply looking at the output numbers is not enough. When dealing with natural language or human ratings, it's safe to assume that there are infinitely many possible instances that could be used for training and testing, of which the ones we actually train and test on are a tiny sample. Thus, it is possible that observed differences in the reported performance are really just noise. There exist statistical methods which can be used to check for consistency (*statistical significance*) in the results, and one of the simplest such tests is the **sign test**. We can now add rigorosity to our answer by appending the following question in conjunction with the original one:

> (Q0.2) Is the performance difference between the two methods *statistically significant*?

Apply the sign test to answer questions (Q0). The sign test is described in Siegel and Castellan (1986)[1], page 80 (scans of the relevant pages are available in the L90 directory). For the purposes of this paired test, treat ties by adding half a point ot either side. When computing significance using the binomial distribution, round up the final two counts to the nearest integer. You can quickly verify the correctness of your sign test code using a free online tool[2]

From now on, report all differences between systems[3] using the sign test. In your report, you should report statistical test results in an appropriate form – if there are several different methods (ie. systems) to compare, tests can only be applied to pairs of them at a time. This creates a triangular matrix of test results in the general case. When reporting these pair-wise differences, you should summarise trends to avoid redundancy.

[1] Siegel and Castellan, Nonparametric Statistics for the behavioural sciences, McGraw-Hill.

[2] For example `http://www.fon.hum.uva.nl/Service/Statistics/Sign_Test.html`

[3] You can think about a change that you apply to one system, as a new system.

1.4 Machine Learning using Bags of Words representations

Your second task is to program a Machine Learning approach that operates on a simple Bag-of-Words (BoW) representation of the text data, as described in Pang et al. (2002). In this approach, the only features we will consider are the words in the text themselves, without bringing in external sources of information. The BoW model is a popular way of representing text information as vectors (or points in space, depending on how you want to look at it), making it easy to apply to classical Machine Learning algorithms on NLP tasks. However, the BoW representation is also very crude, since it discards all information related to word order and grammatical structure in the original text.

1.4.1 Writing your own classifier

Write your own code to implement the Naive Bayes (NB) classifier[4]. As a reminder from the lectures, the NB classifier works according to the following equation:

$$\hat{c} = \underset{c \in C}{\operatorname{argmax}}\, P(c|\vec{f}) = \underset{c \in C}{\operatorname{argmax}}\, P(c) \prod_{i=1}^{n} P(f_i|c)$$

Feature vector $\vec{f}$; most probable class $\hat{c}$; C set of classes.

You may use whichever programming language you prefer (C++, Python, or Java being the most popular ones), but you **must** write the Naive Bayes training and prediction code from scratch. You will **not** be given credit for using off-the-shelf Machine Learning libraries such as **mlpack** (C++), **scikit** (Python), **Weka** (Java), etc.

The data in `/usr/groups/mphil/L90/data` contains the text of the reviews, where each document is one review. Your algorithm should read in and tokenise in the text, store the words and their frequencies in an appropriate data structure that allows for easy computation of the probabilities used in the Naive Bayes algorithm, and make predictions for new instances.

(Q1.0) Train your classifier on files cv000–cv899 from both the `/pos` and the `/neg` directories, and test it on the remaining files cv900-cv999. Report results using simple classification accuracy as your evaluation metric.

(Q1.1) [Optional. Even if you do this, don't report it.] Would you consider accuracy to also be a good way to evaluate your classifier in a situation where 90% of your data instances are of positive movie reviews? Simulate this scenario by keeping the positive reviews data unchanged, but only using **negative** reviews cv000-cv089 for training, and cv900–cv909 for testing. Report the classification accuracy, and explain what changed.

Smoothing

The presence of words in the test dataset that haven't been seen during training can cause probabilities in the Naive Bayes classifier to be 0, thus making that particular test instance undecidable. The standard way to mitigate this effect, as well as to give more clout to rare words, is to use smoothing, in which a probability fraction $\frac{f(w)}{N_{words}}$ for a word w becomes $\frac{f(w)+smoothing(w)}{N_{words}+\sum_{\omega \in V} smoothing(\omega)}$.

(Q2.0) Implement constant feature smoothing ($smoothing(\cdot) = \kappa$, for all words) in your Naive Bayes classifier's code, and report the impact on performance.

[4]This section and the next aim to put you a position to replicate Pang et al., Naive Bayes results. However, the numerical results will differ from theirs, as they used different data.

Crossvalidation

A serious danger in using Machine Learning on small datasets, with many iterations of slightly-different versions of the algorithms is that we end up with Type III errors, also called the "testing hypotheses suggested by the data" errors. This type of errors occur when we make repeated improvements to our classifiers by playing with features and their processing, but we don't get a fresh, never-before seen test dataset every time. Thus, we risk developing a classifier that's better and better on our data, but worse and worse at generalizing to new, never-before seen data.

A simple method to guard against Type III is to use cross-validation. In N-fold cross-validation, we randomly shuffle the data, and divide it into N distinct chunks. Then, we repeat the experiment N times, each time holding out one of the chunks for testing, training our classifier on the remaining $N-1$ data chunks, and reporting performance on the held-out chunk.

> (Q3.0) Write the code to implement 10-fold cross-validation for your Naive Bayes classifier (version (Q5)), and compute the 10 accuracies.

YOU HAVE NOW REACHED THE MINIMAL REQUIREMENT FOR THE BASELINE SYSTEM. ANYTHING BEYOND THIS POINT IS AN EXPLANATION OF PANG ET AL.

1.4.2 Features, overfitting, and the curse of dimensionality

In the Bag-of-Words model, ideally we would like each distinct word in the text to be mapped to its own dimension in the output vector representation. However, real world text is messy, and we need to decide on what we consider to be a word. For example, is "`word`" different from "`Word`", from "`word,`", or from "`words`"? Too strict a definition, and the number of features explodes, while our algorithm fails to learn anything generalizable. Too lax, and we risk destroying our learning signal. In the following section, you will learn about confronting the feature sparsity and the overfitting problems as they occur in NLP classification tasks.

A touch of linguistics

> (Q4.0) Taking this a step further, you can use stemming to hash different inflections of a word to the same feature in the BoW vector space. How does the performance of your classifier change when you use stemming on your training and test datasets[5]?
>
> (Q4.1) Is the difference from the results obtained at (Q1) and (Q2) statistically significant?
>
> (Q4.2) What happens to the number of features (ie. the size of the vocabulary)? Give actual numbers.

Putting [some] word order back in

> (Q5.0) A simple way of retaining some of the word order information when using BoW representations is to use bigrams or trigrams as features. Retrain your classifier from (Q4) using bigrams or trigrams as features, and report accuracy and statistical significance in comparison to the experiment at (Q4).
>
> (Q5.1) How many features does the BoW model have to take into account now? How does this number compare (eg.: linear, square, cubed, exponential) to the number of features at (Q4)?
>
> (Q6.1) Take the best and the worst of the 10 accuracies obtained at (Q3.0). Is the difference between them statistically significant? How do you interpret this result in terms of having overfitted, and committed a Type III error?

[5]Please use the Porter stemming algorithm, available at `http://tartarus.org/martin/PorterStemmer/`

Feature independence, and comparing Naive Bayes with SVM

Though simple to understand, implement, and debug, one major problem with the Naive Bayes classifier is that it takes a performance hit (becomes skewed) when it is being fed features which are not independent (ie. are correlated). Another popular classifier that doesn't scale as well to big data, and is not as simple to debug as Naive Bayes, but that doesn't assume feature independence is the Support Vector Machine (SVM) classifier.

> (Q7.0) Write the code to print out your BoW features from (Q5) in SVM light format[6].
> (Q7.1) Download and use the SVM Light implementation[7] on our dataset. Compare the classification performance of SVM Light to that of the Naive Bayes classifier, and explain the numbers.

More linguistics

Now install the Stanford parser[8] (instructions on separate sheet). It offers POS-tagging as a pre-step to parsing. Use its POS tagger on your texts. Replicate what Pang et al. were doing:

> (Q8.0) Replace your features by word+POS features, and report performance. Does this help? Why?
> (Q8.1) Discard all closed-class words from your data (keep only nouns, verbs, adjectives and adverbs), and report performance. Does this help? Why?

[6]Described in detail on `http://svmlight.joachims.org/`
[7]SVM Light is available for download at `http://svmlight.joachims.org/`
[8]Download from `http://nlp.stanford.edu/software/lex-parser.shtml`